WiFi Gate Guard
A Captive Portal Implementation for Home Networks

Ben Blumenberg
May 21, 2018
Computer Science Department
College of Engineering
California Polytechnic State University
San Luis Obispo

Copyright

Table of Contents

The Problem

Passwords. Used ubiquitously as a means of authentication in the digital world, passwords are dreaded yet unavoidable. The core issue with passwords stems from the fact that anything that's easy for a human to remember is easy for a computer to guess. Users are cautioned against using simple passwords, like birthdays or even dictionary words, lest their passwords be guessed and their data be stolen. When a typical desktop computer can guess passwords on the order of millions of guesses per second, such simple password provide laughably little security. But the answer is not as simple as choosing a long, random password with lots of numbers, letters, and special characters. If data is so secure that it's owner can't remember the password needed to access it, then what's the point of keeping the data at all?

So what to do about the problems associated with password-based authentication? Storing passwords as salted hashes rather than plaintext is one modern technique which vastly improves the implementation of password-based authentication.[1] Without going into detail, this technique means that an attacker, after successfully breaking into a password database, would still need to crack each stored password individually before gaining any useful information. However, even when implemented correctly, this scheme doesn't do *anything* to make a simple password any less guessable.

Password managers like Apple's iCloud Keychain, or the popular web-based alternative LastPass, help with this problem by storing long, random, hard to remember passwords behind one master password.[2] However, the problem remains that the user still must remember the master password, and the scheme is only as secure as the master password is (crack the master password and *all* of the user's other passwords are revealed).

Certain other authentication types, like fingerprints, face scans, or usb-keys are also gaining popularity. While these leverage a different type of authentication (something you are or have, versus something you know), they are often layered on top of or used in conjunction with password-based authentication.[3] While they do provided easier and/or more secure authentication than basic passwords, they still don't solve the underlying issue, that passwords are hard to use.

The above schemes certainly help obscure the issues that arise from password-based authentication, but they are hardly as ubiquitous as passwords themselves are. Home WiFi, for example, doesn't benefit from any of the discussed "solutions" (at least not without a lot of technical know-how and the time to design a customized solution). Most home networks have a single password that is shared by all users, which is problematic for a few reasons. The primary issue is the difficulty inherent in distributing the password, most of which are set to a long, random string by the internet service provider. While secure, accurately reading the password, speaking each character aloud, hearing each character accurately, and typing each in on a device is a trying process at best for both host and guest. Furthermore, the password if often stored in a very insecure location, usually on a sticker on the bottom of the router. And, since the password is shared among all users, the security it provides only keeps data secret from anyone not on the WiFi network, making the authentication scheme functionally an access control scheme as opposed to a fully-fledged security solution. Lastly, sharing the

---

[1] Zachary NJ Peterson (lecture, California Polytechnic State University, San Luis Obispo, January 2018).

[2] Ibid

[3] Ibid

password with everyone who wants to use the network makes keeping it secret for any length of time a virtual impossibility.

The Solution

The idea behind this project is to reinvent the user experience for a host providing WiFi to house guests, and for guests requesting WiFi access while visiting. While hosting a simple open WiFi network is extremely easy, it does not provide *any* security and is thus an unacceptable solution for most use cases. The solution needs to provide some semblance of security, access control at the very least, and ideally serve as a platform on which more secure implementation could be based.

User Experience

The chosen solution uses a captive portal to provide access control on an open network (similar to what is typically used in coffee shops and hotels), integrated with an app which runs on the host's smartphone or tablet. This solution provides several benefits in terms of creating the best possible user experience, by using existing technologies that work with all modern devices, and by not requiring each guest to install or use any proprietary software. Furthermore, once the system is set up and running, any day-to-day administration the host has to do can be handled via a single app.

With the technologies which power the solution chosen, the core user experience (a guest requesting network access) can be outlined:

- The guest user searches for available WiFi networks and chooses (or is verbally directed to by the host) the open guest network
- The guest's device automatically detects that a captive portal is being used and loads the login page
- The guest user enters their name on the login page and requests access
- *The host user receives a notification indicating that someone is requesting WiFi access*
- *The host user opens the app and responds the request, either allowing or denying access based on the name associated with the request*
- The login page on the guest's device automatically continuously checks whether an authentication decision has been made by the host
- The login page on the guest's device either closes automatically if access has been granted, or displays a message asking the user to talk to the host should an error occur or access be denied

Typical captive portal implementations seen in coffee shops and hotels usually either ask users to simply agree to terms and conditions before granting WiFi access, use some sort of automatic backend to authenticate users using usernames a password or room numbers, or ask for payment before access is granted. This solution effectively simplifies both the user's experience logging in, the host's experience administrating access, and still provides better access control than then simple agreement to terms and conditions. Asking for only a name rather than asking users to register with a username and password or type in a credit card number makes requesting access as a guest extremely straightforward and easy, and the ability to manage authentication via a simple app rather than needing to maintain a complex backend database greatly eases the burden on the host.

Implementation

While users and hosts will only ever interact with the authentication webpage and the administration app, WiFi Gate Guard requires several components to make everything work.

The software that powers WiFi Gate Guard can be broken down into four main sections:
- The captive portal software which runs on the home router
- The authentication webpage which serves as the main point-of-contact for guest users
- The backend server which hosts the authentication webpage and links everything together
- The mobile app which serves as the main point-of-contact for the host user

Hardware and Operating Systems

WiFi Gate Guard requires two main pieces of hardware to function:
- A smartphone capable or running the administration app and alerting the user when action is required
- A wireless router capable of running OpenWrt and with enough processing power and storage to:
    - manage the wireless network
    - implement the captive portal
    - keep the network secure using a firewall
    - administrate access to the network
    - communicate with the mobile app
    - serve the authentication webpage

In the development of this project I used devices which I personally own, as I was already intimately familiar with their basic use and operation. This made access to appropriate devices a non-issue and facilitated easy and rapid development. The router I own does not have enough processing power or storage to fulfill all of the functionality listed above, so I offloaded a portion of the functionality onto a Raspberry Pi. To develop, test, and demonstrate the project I used the following hardware and operating systems:
- TP-Link Archer C7 AC1750, a home wireless router running the OpenWrt operating system[4]
- Apple iPhone X, a smartphone running the iOS 11 operating system[5]
- Raspberry Pi 3 (Model B), a miniature computer running the Raspbian operating system[6]

In this development setup the router is responsible for managing the wireless network, implementing the captive portal, and keeping the network secure, while the Raspberry Pi is responsible for administrating access to the captive network, serving the authentication webpage, and facilitating communication between itself, the router, and the app. This three-piece setup is a viable alternative for any setup in which the router is not capable of supplying all of the required functionality.

---

[4] "Welcome to the OpenWrt Project," OpenWrt: Wireless Freedom, , accessed June 10, 2018, https://openwrt.org/.

[5] iOS, computer software, version 11, Apple.com, September 19, 2017, accessed June 10, 2018, https://www.apple.com.

[6] "Raspbian," Raspberry Pi, , accessed June 10, 2018, https://www.raspberrypi.org/documentation/raspbian/.

Choosing the right router software is the key starting point for any home network project. WiFi Gate Guard is built around the OpenWrt linux kernel, an open source kernel built specifically for networking appliances. OpenWrt gives access to a slimmed-down yet fully-functional, on-device, linux operating system which allows direct SSH access, a vast library of installable binaries, and which is designed to incorporate several networking services (such as wired and wireless network access, firewall, captive portal, and VPN) into one cohesive environment. In other words, it is *the* crucial component which enabled development of WiFi Gate Guard.

Captive Portal

This is the portion of the project includes the actual wireless network that clients connect to, as well as the software which tracks clients who connect to the wireless network and redirects unauthenticated clients to the authentication webpage.

In order to operate a captive portal, WiFi Gate Guard relies on a couple important pieces of functionality which are provided by OpenWrt. First, OpenWrt comes pre-configured with the ability to not only manage wireless networks, but also to create one or more virtual networks without the need for additional wireless antennae. Many manufacturer-installed router operating systems provide similar functionality, but most are limited to providing a single additional guest network (per wireless antenna) which may or may not provide basic WPA2 security.[7] With OpenWrt, it is trivial to create several virtual networks on a single antenna, each with its own virtual interfaces, security configuration, firewall rules, DHCP server, etc. Clients will never know or care that there may be more than one wireless network running off of a single antenna. WiFi Gate Guard operates on one of these virtual networks, allowing the home router and pre-existing wireless networks to function as normal alongside the project.

The second important feature provided by OpenWrt is the ability to install third party software packages from a central repository, through a package manager. Nodogsplash, an open source project on GitHub, is one of many packages available for OpenWrt, and provides a simple and lightweight captive portal implementation.[8] Nodogsplash tracks clients by MAC address and groups them into preauthenticated and authenticated groups. When a preauthenticated client attempts an HTTP connection from a web browser, the content of the page is automatically replaced by the content of the authentication webpage. Nodogsplash is configured via text file, and operates on a single wireless interface. The configuration file allows several useful options to be configured. For this project, authentication timeouts were set and additional firewall rules were configured to apply to users in the preauthenticated and authenticated groups. Preauthenticated users are limited to using UPD/TCP port 53 for DNS, and TCP port 5050 at the public IP address of the router (which, via port forwarding set up within OpenWrt, connects to the backend server running on the Raspberry Pi). Authenticated users can also use TCP ports 22, 80, and 443 for SSH, HTTP, and HTTPS respectively. An idle client will be deauthenticated automatically after 24 hours, and an active client will be deauthenticated after a week. These rather relaxed timeouts should be sufficient for a home network, and allow guests staying multiple days to retain connectivity without having to reauthenticate.

---

[7] "How to Configure Guest Network on Dual Band Wireless Routers?" How to Select the Operating Mode of TP-Link Wireless Multiple Modes Devices? - TP-Link, , accessed June 10, 2018, https://www.tp-link.com/us/faq-649.html.

[8] mwarning et al., Nodogsplash, computer software, version 1.0.1, GitHub, December 21, 2016, accessed June 10, 2018, https://github.com/nodogsplash/nodogsplash.

Authentication Webpage

The authentication webpage is the portion of the project which clients connecting to the network see in their device's web browser.

Most modern operating systems perform some sort of automatic check for a captive portal when connecting to an unencrypted network, and will load the authentication webpage immediately after connecting.[9] This is usually done by attempting to connect to a know-good url (Apple devices use captive.apple.com); if the page at the other end of the connection is not what the device expects, then there's probably a captive portal in the way and the authentication page is loaded and displayed to the user. On devices which don't support this or a similar feature, users will get automatically redirected to the authentication webpage the first time they try to load any HTTP webpage. It's worth noting that there are a few limitations to this functionality for devices which don't automatically detect the captive portal. First, if a user attempts to connect to the internet through something that's not a web browser (like an email client), they will be unable to connect and the authentication page won't get loaded. Second, this form of redirection - where the content of the page the user is trying to load is replaced with the content of the authentication page - breaks the SSL/TLS security model.[10] As such HTTPS connections will not be redirected and users will again be unable to connect without ever seeing the authentication page. As more and more of the websites we visit adopt encryption, the chances of this in-browser redirection working get lower and lower. Unfortunately these are limitations of the underlying technologies rather than anything that can be fixed via configuration. Any solution that avoids these limitations would require wide-ranging support from client devices, just as this solution more or less relies on the automatic captive portal detection discussed above.

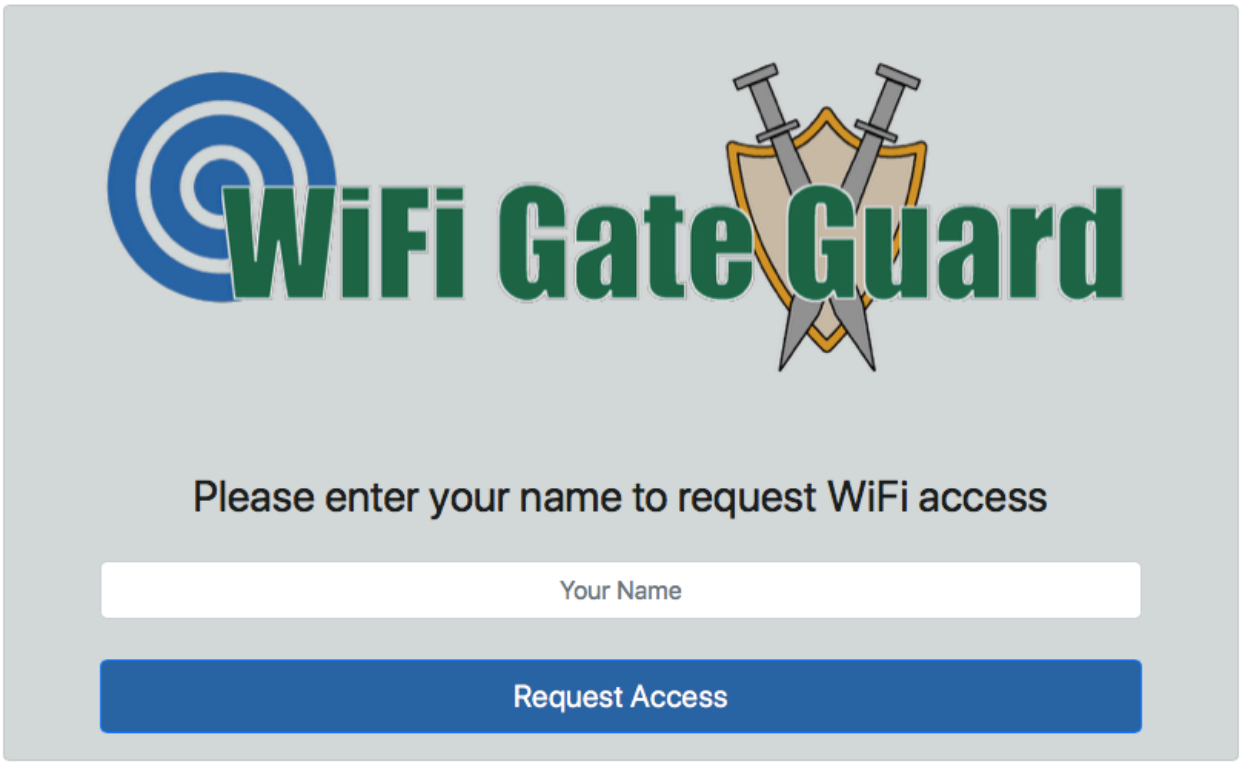
Once the user's device load the authentication page served by nodogsplash they are again redirected to the actual authentication page. This redirection is performed by some html code which was generated to leverage several features built into html, in the hopes that at least one will work for whichever browser is being used. If automatic redirection fails, the user is presented with a clickable link which will take them to the actual authentication page. This redirection allows the actual authentication page to be served by a more powerful device (in this case the Raspberry Pi) and to use TLS encryption to keep user data sent between the webpage and the backend server confidential. As an additional benefit, serving both the authentication webpage and backend server from the same device means they can rolled into one complete server, simplifying the project somewhat. (See the backend server section below for more information about what it does and what software is used to power it.)

---

[9] Glenn Fleishman, "Can You Free Yourself from Captive.apple.com?" Macworld, January 05, 2018, , accessed June 10, 2018, https://www.macworld.com/article/3241896/ios/can-you-free-yourself-from-captiveapplecom.html.

[10] Cjoseph, "Redirect to Captive Portal Using HTTPS," Airheads Community, November 20, 2015, , accessed June 10, 2018, http://community.arubanetworks.com/t5/Wireless-Access/Redirect-to-Captive-Portal-using-HTTPS/td-p/252597.

The actual authentication webpage that clients see looks like this:



The goal behind the design of the authentication webpage was to keep it as simple as possible, to stay in line with the guiding motivation of this project - creating the best possible user experience. Many users are used to seeing captive portal pages show up on their devices when connecting to WiFi at hotels and coffee shops, but not in a home environment. To accomplish the goal of ease and simplicity, the page was designed to be immediately recognizable as a login page and to clearly and simply guide the user towards wireless access. The page outwardly has only two components with which the user interacts: A box where they can enter their name, and a big button to request access. The popular Bootstrap frontend component library was used give the page a polished look, and to keep coloring and component design consistent with what users will be used to from other web pages.[11] A simple, colorful logo which emphasizes wireless access and security adds to the polish of the page, making it much more approachable to all users.

Once the user enters his or her name and clicks the "Request Access" button the page uses javascript to dynamically submit the request to the backend server, before displaying a loading animation to keep the user confident that the page is doing something.[12] The page then continuously polls the backend server for an authentication decision with an initial 10 second delay, then a 5 second delay between subsequent checks (to maintain a balance between server load and wait-time for users). Once an authentication decision is received the page

[11] Mdo and Fat, Bootstrap, computer software, version 4.1.1, Bootstrap, April 30, 2018, accessed June 10, 2018, getbootstrap.com.

[12] Javascript, computer software, JavaScript.com, 2018, accessed June 10, 2018, https://www.javascript.com.

takes the appropriate action depending on whether the request was approved: If access was granted, the user is automatically redirected to the original page they requested. If access was denied, a message is displayed to the user instructing them to ask the host for more information.

Backend Server

The backend server is the portion of the project which serves the authentication webpage, facilitates communication between the webpage and the app, and tracks open requests for wireless access.

The server is run on the Raspberry Pi and is built on the Flask python framework.[13] Flask is a simple and easy-to-use framework which allows the python programming language to be used to create RESTful web service.[14] REST (or ReST) stands for Representational State Transfer;[15] in practice a RESTful service is just a web service which provides a number of functions accessible through HTTP requests such as GET and POST requests. With Flask, several "routes" can be defined each of which maps a string and a list of HTTP methods to a python function. For example, when a user makes an HTTP GET request to myFlaskServer.com/hello_world, the function assigned to the route "/hello_world" gets executed (assuming such a route and function exist, and the route accepts GET requests). Each function returns some sort of response data, which might be JSON formatted data, an HTML webpage, a simple string, or just about anything else, along with an HTTP status code (i.e. 200 for success[16]). The response data and status code get sent back to the requester.

Flask provides a number of more advanced features, and is open-source, allowing third parties to add additional functionality. One such advanced feature that WiFi Gate Guard makes use of is the ability to capture variables in the route definition. Routes can capture several basic data types, such as strings and integers using the following format: "/myRoute/<varType:varName>". The function associated with the route must take the variables declared in the route as parameters. A function assigned to "/accessRequest/<string:name>" would need to take a string called name as its only parameter. This functionality allows data to be easily passed to the flask application, allowing for the functionality required by WiFi Gate Guard.

---

[13] Armin Ronacher, Flask, computer software, version 1.0.2, Flask, 2010, accessed June 10, 2018, http://flask.pocoo.org.

[14] Python Software Foundation, Python, version 2.7, Python, 2018, accessed June 10, 2018, https://www.python.org.

[15] Sam Deering, "Do You Know What a REST API Is?" SitePoint, December 07, 2012, accessed June 10, 2018, https://www.sitepoint.com/developers-rest-api/.

[16] Camslice, "HTTP Response Status Codes," MDN Web Docs, June 5, 2018, accessed June 10, 2018, https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

The server uses a simple JSON-based database, powered by pickleDB[17] and SimpleJSON[18], to track open access requests. Each request is stored with a unique identifier (to avoid conflicts between requests with the same names), the name on the request, and a timestamp indicating when the request was made. Another python script, separate from the main script which runs the Flask server, is periodically activated by cron (a unix tool which runs arbitrary shell commands at defined intervals) to clean out stale requests, keeping the database as small as possible.[19] Whenever the database file is opened (whether by the server or the cleaner) it is locked to avoid simultaneous access issues.

The backend server exposes seven routes who's functionality is described below:

Route	Functionality
/open	This is used by the app to retrieve a list of requests (from the database) for which an authentication decision has not yet been made.
/req/<string:name> **Ex: /req/Ben**	This is the primary entry point for the backend server, and is used by the authentication webpage to submit a new access request, where 'name' is the name entered by the user on the authentication page. When this endpoint is activated a new request is entered into the database, with a new UUID, 'name', and a timestamp. A push notification is then sent to the app, requesting an authentication decision (see the Communication section below for more detail on how this is accomplished). The UUID from the new request is returned to the requester to facilitate polling.
/auth **/<string:requestId>** **/<string:decision>** **Ex: /auth/123/true**	This route is how the app submits an authentication decision. 'requestId' is a UUID which should match the id of an open request, and 'decision' should be "true" to grant access or anything else to deny access. When this endpoint is activated the decision is stored as a boolean value in the database. If the id is not found, a 404 NOT FOUND status code is returned, otherwise 200 SUCCESS is returned. No data is returned (an empty string).
/auth/ **/<string:requestId>** **Ex: /auth/123**	This is used by the authentication webpage to poll for an authorization decision. 'requestId' is a UUID which should match the id of a request in the database. If the id is not found, a 404 NOT FOUND status code is returned, indicating that the requestId is invalid or the request has timed out. If the id is found and access has been granted, 200 SUCCESS is returned. If access has been denied, 403 UNAUTHORIZED is returned. If no authentication decision has been recorded, 202 ACCEPTED is returned, indicating that the page should continue polling. The authorization webpage uses these return codes to display appropriate information to the user (or continue displaying the loading animation as long as polling continues). When either a 200 or 403 is returned (the request exists, a decision has been recorded, and the decision has been sent back to the webpage) the corresponding request is removed from the database.

[17] Harrison Erd, PickleDB, computer software, version 0.6.2, PickleDB, February 24, 2016, accessed June 10, 2018, https://pythonhosted.org/pickleDB/.

[18] Bob Ippolito, Simplejson, computer software, version 3.14.0, PyPi, April 21, 2018, accessed June 10, 2018, https://pypi.org/project/simplejson/.

[19] Paul Vixie, Cron, computer software (2010).

Route	Functionality
/registerAPNToken /<string:apnToken>	This route is used by the app to register for push notifications. See the Communication section below for more information on how push notifications work.
/portal/<string:tok>	This route returns the authentication webpage, which is displayed to the user. 'tok' is the nodogsplash authentication token (see the Security section below for information on what this is and how it is used).
/p/<path:path>	This route allows the authentication webpage to access separate CSS (formatting) and javascript (code) files necessary for its operation. Separating these files from the main HTML (formatting) file allows the page to load more quickly.

Flask includes a lightweight server (which uses the python code discussed above to actually respond to web requests) for development, but for production use a more robust server is required. WiFi Gate Guard uses the gunicorn[20] server, and manages it via supervisord[21]. Supervisord is a unix program which can automatically run and monitor other unix programs, in this case gunicorn. By adding a line to /etc/rc.local on the Raspberry Pi, supervisord is automatically started at boot. Supervisord itself is configured to automatically start gunicorn, and restart it should it fail. Any errors (or other output) from gunicorn is saved to a log file for debugging, should that be necessary. With supervisord configured properly, the normal operation of the backend server requires no user interaction or maintenance whatsoever.

Gunicorn is a python web server compatible with the Flask framework which takes a flask app and runs it as a persistent and multi-threaded server. To keep things simple, and because the benefit from multithreading would be minimal given its design, WiFi Gate Guard uses only a single thread for the backend server. Gunicorn can be configured to run on a specific port, on a specific IP (which should be an IP assigned to the device), as well as use a TLS certificate and private key to provide encrypted access via HTTPS. WiFi Gate Guard uses gunicorn to provide encrypted access to the authentication webpage as well as the rest of the functionality provided by the backend server.

TLS certificates can be expensive, but WiFi Gate Guard makes use of a free certificate from Let's Encrypt rather than paying for one from companies like GoDaddy or DigiCert.[22] Let's Encrypt is a completely free and totally automated certificate authority. Using the certbot utility users with shell access on their servers can obtain a TLS certificate and private key in a matter of minutes.[23] All that's required is that the user allows certbot to create and run a temporary web server on the address for which the certificate was requested, and that the address in question is accessible from the internet (i.e. access is not limited to a private local area network). Using certbot and its temporary server, Let's Encrypt can connect to the server from the outside internet, verifying that the requester has administrative control over the device at

---

[20] Benoit Chesneau, Gunicorn, computer software, version 19.8.1, Gunicorn, April 30, 2018, accessed June 10, 2018, http://gunicorn.org.

[21] Agendaless Consulting, Supervisord, computer software, version 3.3.4, Supervisor, February 15, 2018, accessed June 10, 2018, http://supervisord.org/index.html.

[22] "Free SSL/TLS Certificates," Let's Encrypt, accessed June 10, 2018, https://letsencrypt.org/.

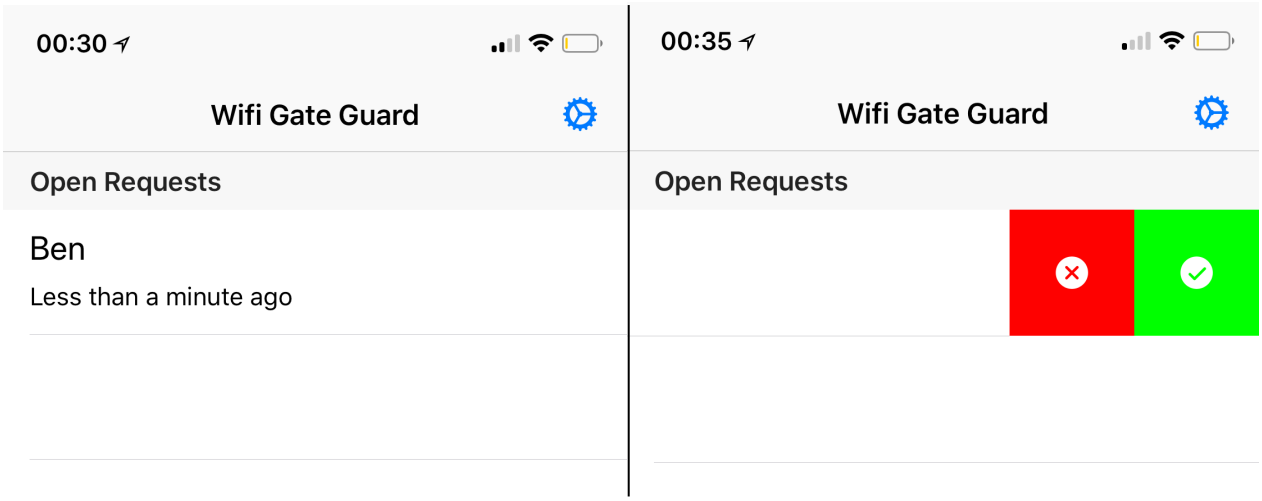
[23] Certbot-auto, computer software, Certbot, accessed June 10, 2018, https://certbot.eff.org/lets-encrypt/pip-other.

the address in question. That's enough grounds for Let's Encrypt to issue a TLS certificate, generated from a certificate signing request which certbot itself generates and submits. The entire process is quick, easy, and surprisingly very automatic.

App

The app is the portion of the project which allows the host to approve or deny requests for network access.

Written in Swift and developed for iOS using Xcode, the app is simple and easy to use.[24] When launched, the app retrieves a list of outstanding requests from the backend server and displays them to a user in a table view (left image below). Swiping to the left on an open request allows the user to approve the request using the green button, or deny the request using the red button (right image below). Tapping either option sends the decision to the backend server and removes the row from the table.



The app maintains a record of open requests as a globally accessible list using the singleton pattern, and periodically checks with the server to keep the list up to date. The periodic checking only happens while the app is open, no quicker than every 15 seconds. The user can also pull down on the table to refresh (a ubiquitous action within iOS), which resets the automatic refresh timer.

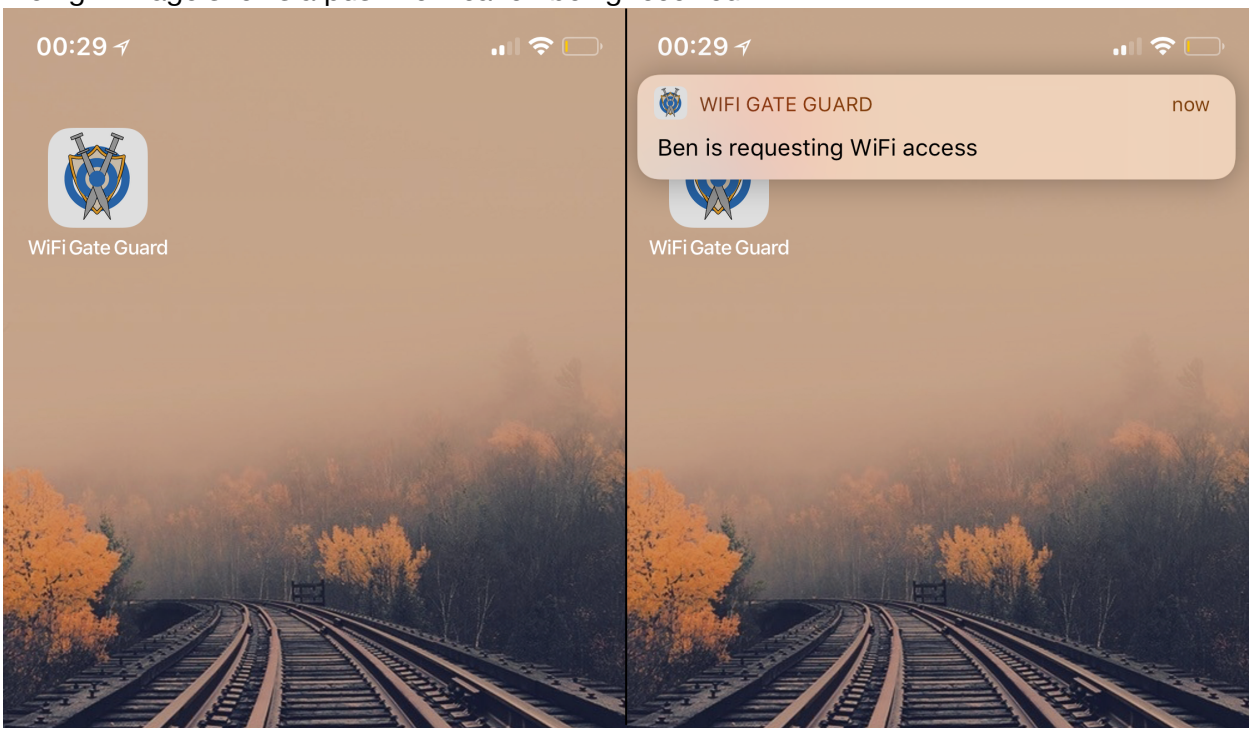
One key decision during the development of the app was where to put the buttons to make a decision for a request. One option was to include them in the row itself, meaning they would always be in view. While this would have made it easier for users to figure out what to do, it would also have cluttered the screen (especially with many simultaneous requests), could have caused longer names to be truncated, and could have been easier to tap accidentally. Hiding the buttons behind a swipe gesture eliminates those issues, but does make the buttons harder to find for users less familiar with iOS. To alleviate that issue, an animation was designed which causes the buttons to "peek" out from the side (and then disappear again) when a row is tapped, hinting to the user that they can swipe over to see available actions.

---

[24] Apple Inc., Xcode, computer software, version 9.4 (9F1027a), Developer, 2018, accessed June 10, 2018, https://developer.apple.com/develop/.

When the app is opened for the first time on a device, it asks the user for permission to display notifications, and registers with the Apple Push Notification service.[25] Once registered, the device receives a unique token which can be used to send it notifications. The app stores the token locally, and checks for its existence and validity whenever the app launches. Whenever the token changes (or it is generated for the first time) the app sends this token to the backend server. By tapping the gear icon in the upper right corner of the app, the user can access the settings view, where they can manually ask the app to resend this token to the backend server (useful if the backend database is unexpectedly wiped). Whenever the app receives a push notification it immediately refreshes its list of outstanding requests, ensuring that by the time the app is visible the list of open requests is current.

While functionality and usability of the app are the primary concerns, design also plays a large role in the success of an app. Stock iOS designs are used wherever possible, to keep the app familiar to users who are used to the operating system. Furthermore, the approve and deny buttons are simplified via color coding and simple, easy-to-understand glyphs. The icon designed for the app incorporates elements from the logo displayed on the login page to give the project visual cohesiveness. The left image below shows the app icon on the home screen, the right image shows a push notification being received:



Communication Between Components

The bulk of the projects functionality is covered in the above sections. This sections serves primarily to summarize the flow of information through different parts of the project.

Information starts in the authentication webpage when a client requests access. The webpage uses a synchronous XHTTP request to submit the request to the backend server, and retrieve

[25] Apple Inc., "UserNotifications," Developer, 2018, accessed June 10, 2018, https://developer.apple.com/documentation/usernotifications.

the id for the new request.[26] The webpage again uses synchronous XHTTP requests to continuously poll the backend server for an authentication decision.

Once the backend server creates a new request and stores it in the database, it sends a push notification to the app via the Apple Push Notification service. This communication leverages the HTTP/2 protocol and JSON Web Tokens (jwt) to implement token-based authentication.[27] Using jwt, a token is created that includes the Team ID of the app's developer along with a timestamp for the request. The token is then encrypted using a private key, and headers are prepended to the token which specify the encryption algorithm used and a key id referencing the private key. This token is sent along with other headers which include information such as identifying information for the app, as well as the body of the request containing the message to be displayed in the notification, the name of the sound which should be played, and a number that should be as an app badge (in this case indicating the number of outstanding requests). The device token (which the app must send to the backend server) is included in the request url, letting the Notification service know which device to send the notification to. This whole process is accomplished in python on the backend server using the Hyper HTTP/2 Client[28] and PyJWT[29].

The app uses several of the rest endpoints made available on the backend server to both retrieve and send information. When the app registers for push notification immediately after startup it sends its device token to the backend server. It then retrieves a list open requests to display to the user. When the user makes an authentication decision, this decision is sent to the backend server, again via one of the rest endpoints. The backend server then stores the decision in the database and sends it to the authentication webpage the next time a polling request is received.

Since nodogsplash by default blocks all internet communication besides DNS and connections to its authentication webpage for preauthenticated users, the configuration must be modified to allow connections to the backend server for users to be able to see and interact with the actual authentication webpage hosted there.

It is also worth noting that the backend server is available from anywhere on the web and is accessed via the router's public IP address and port forwarding, meaning that the server itself doesn't actually have to be on the same network or even in the same place as the router running Wifi Gate Guard. Furthermore users will be able to use the app the manage wireless access no matter where they are, so long as they have an internet connection.

---

[26] Vitaly-zdanevich, "Synchronous and Asynchronous Requests," MDN Web Docs, March 1, 2018, , accessed June 10, 2018, https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests.

[27] "Local and Remote Notification Programming Guide: Communicating with APNs," Developer - Documentation Archive, June 04, 2018, , accessed June 10, 2018, https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CommunicatingwithAPNs.html#//apple_ref/doc/uid/TP40008194-CH11-SW1.

[28] Cory Benfield, Hyper, computer software, version 0.7.0, PyPi, September 27, 2016, accessed June 10, 2018, https://pypi.org/project/hyper.

[29] Jose Padilla, PyJWT, computer software, version 1.6.1, PyPi, March 18, 2018, accessed June 10, 2018, https://pypi.org/project/PyJWT.

Security

Security is a key aspect of WiFi Gate Guard. Without at least some security improvement over an open network the project would be worthless no matter how great a user experience it might provide. Fortunately, WiFi Gate Guard provides multiple benefits over both open networks and traditional WPA2 secured networks, and provides a basis for future security improvements to be made.

The primary advantage that WiFi Gate Guard provides over an open wireless network is access control. With an open network the only requirements to connect are having a wireless device and being within range of the router or access point. With WiFi Gate Guard every user who wishes to use the network must first request and receive access, and every request is approved or denied by the host directly. With WiFi Gate Guard, the only people on the network are the people who have been explicitly allowed access (assuming no malicious users break in).

The main advantage that WiFi Gate Guard provides over a typical WPA2 secured wireless network is the lack of any sort of password that must be remembered and shared. Sharing a password with others or writing it down in order to remember it opens up additional avenues to gaining network access without direct access from the host. This means that even an attacker with no technical knowledge could easily circumvent the access control provided by WPA2. While WiFi Gate Guard is not a perfectly secure system, it at least requires some technical knowledge to exploit. Furthermore, as discussed a the beginning of the paper, passwords are just plain annoying.

While the project does provide some important benefits over both open networks and WPA2 networks, it's not without its security flaws. Let's take a look at some of the vulnerabilities in WiFi Gate Guard and how they might be exploited. The first big vulnerability is that WiFi Gate Guard expects that users will enter their real name on the authentication webpage. If two people, one an invited guest and another uninvited both request access under the same name, there's no built-in way to determine who is who. For that matter, there's no built-in way to verify that any request actually originated from the person who's name is on the request. In it's present incarnation, the best way around this would be for the host to talk the guest in question directly, and perhaps ask them to enter a different name on the request. Far from a perfect solution.

The second glaring vulnerability in WiFi Gate Guard comes from the way nodogsplash transitions users from the preauthenticated to authenticated states. In order for a user to be authenticated, they must send an HTTP GET request to the authentication url provided by nodogsplash, and must submit the correct authentication token with the request. This scheme is not inherently insecure, but the way it is implemented makes WiFi Gate Guards entire authorization process easily circumventable with a browser's development console and a limited knowledge of javascript. The astute reader may have noticed that in the earlier discussion of the backend server's rest endpoints, a token is passed along (as part of the url) with a request for the portal page. The token is then embedded into the javascript of the page using a simple string replace before the page is served to the user. This token is in fact the authentication token created by nodogsplash, and is all the user needs (along with the authentication url which is statically coded within the same javascript) to give themselves access, without ever submitting an access request.

(Note: It may seem like an obvious solution to this vulnerability would be to not embed the authentication token in the authentication page served by the backend server. The server already sees the token during this process, so why not have the server store the token and only

hand it over to the authentication page, and the user, once the request has been approved? This would certainly make this vulnerability harder to exploit, but its not a perfect solution. Regardless of whether or not the backend server embeds the token in the authentication page that it serves, the token is still embedded in the authentication page that nodogsplash serves (which redirects to the backend server's authentication page). While this method would further obscure the token, it would still be circumventable with marginally more technical know-how. See the Future Work section below for a more complete solution to this problem.)

Lastly, since nodogsplash uses MAC addresses to track clients, WiFi Gate Guard is somewhat vulnerable to MAC address spoofing. However, two active devices on the same network which share a MAC address will not work properly, so it would be difficult to exploit this vulnerability without being discovered.

What about encryption? All WiFi Gate Guard does is add access control to an open network; no wireless encryption is provided, like it is with WPA2. This has two main drawbacks, both of which are relatively unimportant in the majority of circumstances: First, the lack of encryption means that the access control provided by WiFi Gate Guard is actually less secure than that provided by WPA2 (assuming WPA2's password can be kept secret). Using libpcap, the primary C library used for any application which captures network traffic, it is possible to put some wireless cards into "monitor" mode.[30] In monitor mode the card will capture any and all wireless traffic it sees, regardless of whether or not its associated with the network on which the traffic is being transmitted. When used to try to gain information from a WPA2 secured wireless network, all of the captured traffic would be encrypted and therefore relatively useless. Since wireless traffic under WiFi Gate Guard is sent in the clear, libpcap's monitor mode could effectively break its access control. Second, any user who is able to associate with the network WiFi Gate Guard is protecting will be able to see all traffic with or without using monitor mode. With WPA2 every clients traffic is encrypted with a unique key, so data *should* be confidential even from users who know the password and are on the network. In actuality however, each unique key is derived from the shared key and all other information necessary to break WPA2's encryption model is transmitted in the clear, meaning that WPA2 is only slightly better than an unencrypted network in this manner (it requires slightly more technical knowledge to exploit).[31]

So WPA2 provides encryption that's easy to circumvent (assuming an attacker can learn the password), and WiFi Gate Guard provides none whatsoever. Does it matter? Not really. In today's world most websites are using TLS encryption between the client device and the website's server. While TLS has its own set of security problems (which are beyond the scope of this paper), using a TLS-secured site on an open network (or a network protected by WiFi Gate Guard) has no disadvantages over using it on a WPA2 secured network.[32] Furthermore, for anyone to actually intercept wireless traffic, they have to be physically in range of the wireless router. An uninvited guest hiding in the bushes in the front yard is likely to be a lot more obvious than someone enjoying a latte at the nearest Starbucks while snooping your banking activity. As it turns out, wireless encryption doesn't actually add very much security and so isn't strictly speaking necessary, especially if a more robust access control system is in place.

---

[30] Van Jacobson et al., Libpcap, computer software (2015).

[31] Chris Hoffman, "Warning: Encrypted WPA2 Wi-Fi Networks Are Still Vulnerable to Snooping," How-To Geek, December 03, 2014, , accessed June 10, 2018, https://www.howtogeek.com/204335/warning-encrypted-wpa2-wi-fi-networks-are-still-vulnerable-to-snooping/.

[32] Waiwai933, "Is Visiting HTTPS Websites on a Public Hotspot Secure?" Information Security Stack Exchange, January 9, 2011, , accessed June 10, 2018, https://security.stackexchange.com/a/1527.

In conclusion, while WiFi Gate Guard - in its current state - is far from a complete security solution, it does provide a tangible benefit over an open network and is at least on par with a WPA2 secured wireless network in terms of the effort and technical knowledge required to overcome the security it provides. The next section will outline some modifications which could be made to WiFi Gate Guard to provide a more complete security solution.

Future Work

WiFi Gate Guard could be improved in several ways:

- *Stop embedding the nodogsplash authentication token on the authentication webpage.* This is the most exploitable vulnerability in WiFi Gate Guard and could be fixed relatively easily. All nodogsplash does to put the token on its authentication page is a simple string replace, which the backend server then repeats for the actual authentication page. Since nodogsplash is open source, it would be fairly simple to modify its code to instead send the token directly to the backend server. The backend server would then only reveal the token to the authentication webpage once it had received approval from the app. The hardest part of making this modification would be recompiling the modified nodogsplash code to run on OpenWrt. Code must be packaged and run through a cross-compiler on a separate linux build system before it can be installed on a router. While harder than a typical compilation process for a small C application, this process is well documented on the OpenWrt wiki and is fairly easy to achieve.[33]
- *Allow the app to revoke network access.* This would be a welcome feature; in its current iteration, WiFi Gate Guard requires the host to ssh into the router in order to revoke authentication. The biggest hurdle in making this modification is the fact that nodogsplash doesn't provide an easy way to revoke someone else's authentication without either spoofing their MAC address and sending a request to the special deauthentication url (a hack that should not be used in a production system) or via the command line over an ssh connection. The best way to add this feature would probably be to add some sort of server to nodogsplash which could respond to request for currently associated clients, and requests to deauthenticate clients. This would entail recompiling nodogsplash as discussed above, but it could also potentially increase the size of the compiled executable and the processing power required to run it to the point where it became unusable on most home routers. Another option would be to allow the backend server to automatically ssh into the router and deauthenticate clients that way. This is definitely doable with python, but would be much more error prone than a solution built into nodogsplash itself.
- *Allow the app the submit customized timeouts with each approval.* As it currently stands all WiFi Gate Guard clients are subject to the same idle and non-idle timeouts (as discussed earlier in the paper). If timeouts were customizable on a per user basis and via the app, the default timeouts could be lowered significantly and only increased for guests known to be staying for longer durations. Nodogsplash currently provides no way to do this, let alone an easy way, so finding a way to implement this modification would run into similar challenges as the previous modification.
- *Allow the app to dynamically provide additional security questions besides "What is your name?" to appear on the authentication webpage.* This should increase the security of WiFi Gate Guard by making it harder for an uninvited guest to gain access by impersonating an invited guest (just by using their name). Questions should ask for something that only invited guests would know, like the name of the family dog and/or the main course to be served with dinner. Since this modification wouldn't touch nodogsplash at all it would be much easier to

[33] Valentt, "OpenWrt Wiki," OpenWrt Build System – Usage, January 8, 2018, , accessed June 10, 2018, https://wiki.openwrt.org/doc/howto/build.

implement. The app would need to have some additional menu in which questions (and possibly appropriate responses) could be entered. The questions would then be sent to and stored on the backend server. When the authentication webpage loads, it would check with the backend server to see if it should append any additional questions, each with its own text box, to the form. This would require a bit of javascript work to get right, but would definitely be achievable. The responses would be transmitted to the backend server along with the client's name. The backend server could possibly check the responses against pre-approved answers and auto-deny any request not matching approved answers, or just send everything to the app and let the host manually verify each request and its corresponding question answers.

- *Wireless encryption.* As discussed above the usefulness of this feature is marginal at best, but it would still be welcome if it can be implemented in a user-friendly way. There are two usable options for this modification: One would be to use a RADIUS server with something like EAP-TTLS encryption (the details of how this works is beyond the scope of this paper, suffice to say it would provide encryption without needing a password from the user and without the need to install certificates on client devices).[34] This would likely require using a different captive portal solution from nodogsplash which supports the use of RADIUS servers for authentication. Another option would be to spin up additional virtual networks on the router for every client. Clients, once they authenticate, could be asked if they wanted secure access. For those who say yes, a new wireless network could be automatically created, using WPA2 security, with a random password. The name of the new network and the password would then be given to the user via the authentication webpage, and they could manually connect to the new network. The network's subnet would be limited to one usable IP address and could even leverage MAC address filtering, meaning that the network would only be usable by that specific device (unless the MAC address was spoofed). Such a solution would require significantly more steps for the user to get online, but so long as users are given the option to stay on the open network it could be worth it for secure access, and would still probably be easier than having to ask the host for the WiFi password as with a normal WPA2 network. As an additional benefit, such a scheme where every user has his or her own unique secure network and password would avoid many of the pitfalls of a standard shared WPA2 network. It would be interesting to see how many such virtual networks a typical home router could handle without starting to noticeably slow down, and whether or not encryption would interfere with the captive portal detection built into most modern operating systems (not likely but possible, if the check for a captive portal is conditional on the network being unsecured). If operating systems were to evolve to the point where network information could be vended automatically, such that a user's device could be (with permission) automatically switched to the new secure network, this scheme would become much more usable.

Testing

WiFi Gate Guard was tested simply to make sure that the concept is viable and that the implementation works in a controlled environment. The testing methodology used WiFi Gate Guard running on the devices mentioned elsewhere in the paper, and an Apple iPad as the guest device. To complete the test, I attempted to connect to the captive network using the iPad, at which point the authentication page was automatically displayed, after a short delay. I entered my name on the authentication page on the iPad and tapped to request access. Within seconds my iPhone received a notification indicating that "Ben is requesting WiFi access."

---

[34] "802.1X Overview and EAP Types," Intel, March 8, 2018, , accessed June 10, 2018, https://www.intel.com/content/www/us/en/support/articles/000006999/network-and-i-o/wireless-networking.html.

Tapping the notification opened the WiFi Gate Guard app, and I was able to see the request with my name on it displayed. Tapping on the request played the "peek" animation, hinting that I could swipe left to see available actions. Swiping left revealed the approve and deny buttons. I tapped approve, and the request disappeared. After a few second delay, my iPad was granted network access. I browsed to google.com to make sure that I had internet access, which I did.

To test denying access, I manually deauthenticated the iPad via an ssh connection to the router. I then repeated the same steps as above, and tapped deny instead of approve. A message was displayed on the iPad saying that access had been denied and that I should ask my host for more information. I attempted to load google.com anyway, but could not (as expected.)

Challenges

I encountered several challenges during the development of WiFi Gate Guard. Some of these challenges and their solutions are listed here:

- Most TLS certificate providers, including Let's Encrypt, won't provide certificates for raw IP addresses, only for host names. To get around this limitation and get a certificate for the backend server, I setup a dynamic domain name to point to the public IP of my router using the free dynamic domain name resolution service (ddns) provided by No-IP. I then temporarily forwarded ports 80 and 443 from my router to the backend server, and was able to use certbot to obtain a certificate. The certificate is valid for the domain regardless of the port used, so I then changed the backend server to use a nonstandard port and changed my port forwarding rules to only forward the chosen port to the backend server, freeing up ports 80 and 443 for access to the router's web administration page again.
- Nodogsplash worked fine in basic testing with its stock authentication webpage, but as soon as I added the large files needed by Bootstrap it stopped working. It was unclear whether nodogsplash was unable to handle such a large file or if captive portal detection built into iOS gave up after waiting too long for the page to load, but the authentication page would no longer automatically show up when connecting to the wireless network. Offloading the authentication page onto the backend server and redirecting to it from the page which nodogsplash serves effectively solved this issue.
- I tried a couple other captive portal software packages besides nodogsplash in the hopes of finding one which would support wireless encryption using RADIUS and EAP. CoovaChilli is a popular open source captive portal solution which supports many more advanced features than nodogsplash.[35] Unfortunately, despite trying many release versions of the software and several "supported" configurations, I was unable to get CoovaChilli to successfully start on OpenWrt. I debating running CoovaChilli on the Raspberry Pi instead, and routing all traffic through the Pi. While I think I could have made this work, I ended up abandoning the idea. While the Pi's processor is faster than that of the router, its network card is not meant to handle an entire network's traffic, as the router's is. Making this work would have required significant modifications to how my existing network was set up and would have likely resulted in unacceptable slowdowns. Given the relative little value of encryption, and the possibility of an encryption scheme that could be built on top of nodogsplash (as discussed in the Future Work section above) I decided to stay with the simple, easy-to-use, and (most importantly) functional nodogsplash.

[35] CoovaChilli, computer software, Coova.org, accessed June 10, 2018, https://coova.github.io.

Works Referenced

"802.1X Overview and EAP Types." Intel. March 8, 2018. Accessed June 10, 2018. https://www.intel.com/content/www/us/en/support/articles/000006999/network-and-i-o/wireless-networking.html.

Agendaless Consulting. Supervisord. Computer software. Version 3.3.4. Supervisor. February 15, 2018. Accessed June 10, 2018. http://supervisord.org/index.html.

Apple Inc. Xcode. Computer software. Version 9.4 (9F1027a). Developer. 2018. Accessed June 10, 2018. https://developer.apple.com/develop/.

Apple Inc. "UserNotifications." Developer. 2018. Accessed June 10, 2018. https://developer.apple.com/documentation/usernotifications.

Benfield, Cory. Hyper. Computer software. Version 0.7.0. PyPi. September 27, 2016. Accessed June 10, 2018. https://pypi.org/project/hyper.

Camslice. "HTTP Response Status Codes." MDN Web Docs. June 5, 2018. Accessed June 10, 2018. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

Certbot-auto. Computer software. Certbot. Accessed June 10, 2018. https://certbot.eff.org/lets-encrypt/pip-other.

Cjoseph. "Redirect to Captive Portal Using HTTPS." Airheads Community. November 20, 2015. Accessed June 10, 2018. http://community.arubanetworks.com/t5/Wireless-Access/Redirect-to-Captive-Portal-using-HTTPS/td-p/252597.

CoovaChilli. Computer software. Coova.org. Accessed June 10, 2018. https://coova.github.io.

Deering, Sam. "Do You Know What a REST API Is?" SitePoint. December 07, 2012. Accessed June 10, 2018. https://www.sitepoint.com/developers-rest-api/.

Erd, Harrison. PickleDB. Computer software. Version 0.6.2. PickleDB. February 24, 2016. Accessed June 10, 2018. https://pythonhosted.org/pickleDB/.

Fleishman, Glenn. "Can You Free Yourself from Captive.apple.com?" Macworld. January 05, 2018. Accessed June 10, 2018. https://www.macworld.com/article/3241896/ios/can-you-free-yourself-from-captiveapplecom.html.

"Free SSL/TLS Certificates." Let's Encrypt. Accessed June 10, 2018. https://letsencrypt.org/.

Hoffman, Chris. "Warning: Encrypted WPA2 Wi-Fi Networks Are Still Vulnerable to Snooping." How-To Geek. December 03, 2014. Accessed June 10, 2018. https://www.howtogeek.com/204335/warning-encrypted-wpa2-wi-fi-networks-are-still-vulnerable-to-snooping/.

"How to Configure Guest Network on Dual Band Wireless Routers?" How to Select the Operating Mode of TP-Link Wireless Multiple Modes Devices? - TP-Link. Accessed June 10, 2018. https://www.tp-link.com/us/faq-649.html.

IOS. Computer software. Version 11. Apple.com. September 19, 2017. Accessed June 10, 2018. https://www.apple.com.

Ippolito, Bob. Simplejson. Computer software. Version 3.14.0. PyPi. April 21, 2018. Accessed June 10, 2018. https://pypi.org/project/simplejson/.

Jacobson, Van, Craig Leres, Steven McCanne, and Lawrence Berkeley National Laboratory, University of California, Berkeley. Libpcap. Computer software. 2015.

Javascript. Computer software. JavaScript.com. 2018. Accessed June 10, 2018. https://www.javascript.com.

"Local and Remote Notification Programming Guide: Communicating with APNs." Developer - Documentation Archive. June 04, 2018. Accessed June 10, 2018. https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CommunicatingwithAPNs.html#//apple_ref/doc/uid/TP40008194-CH11-SW1.

Mdo, and Fat. Bootstrap. Computer software. Version 4.1.1. Bootstrap. April 30, 2018. Accessed June 10, 2018. getbootstrap.com.

Mwarning, Lynxis, Bluewavenet, Sayuan, Blogcin, Smoe, Champtar, et al. Nodogsplash. Computer software. Version 1.0.1. GitHub. December 21, 2016. Accessed June 10, 2018. https://github.com/nodogsplash/nodogsplash.

Padilla, Jose. PyJWT. Computer software. Version 1.6.1. PyPi. March 18, 2018. Accessed June 10, 2018. https://pypi.org/project/PyJWT.

Peterson, Zachary NJ. Lecture, California Polytechnic State University, San Luis Obispo, January 2018.

Python Software Foundation. Python. Version 2.7. Python. 2018. Accessed June 10, 2018. https://www.python.org.

"Raspbian." Raspberry Pi. Accessed June 10, 2018. https://www.raspberrypi.org/documentation/raspbian/.

Ronacher, Armin. Flask. Computer software. Version 1.0.2. Flask. 2010. Accessed June 10, 2018. http://flask.pocoo.org.

Chesneau, Benoit. Gunicorn. Computer software. Version 19.8.1. Gunicorn. April 30, 2018. Accessed June 10, 2018. http://gunicorn.org.

Valentt. "OpenWrt Wiki." OpenWrt Build System – Usage. January 8, 2018. Accessed June 10, 2018. https://wiki.openwrt.org/doc/howto/build.

Vitaly-zdanevich. "Synchronous and Asynchronous Requests." MDN Web Docs. March 1, 2018. Accessed June 10, 2018. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests.

Vixie, Paul. Cron. Computer software. 2010.

Waiwai933. "Is Visiting HTTPS Websites on a Public Hotspot Secure?" Information Security Stack Exchange. January 9, 2011. Accessed June 10, 2018. https://security.stackexchange.com/a/1527.

"Welcome to the OpenWrt Project." OpenWrt: Wireless Freedom. Accessed June 10, 2018. https://openwrt.org/.

Works Consulted

Grinberg, Miguel. "Running Your Flask Application Over HTTPS." Miguelgrinberg.com. June 3, 2017. Accessed June 10, 2018. https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https.

"HTML." W3Schools Online Web Tutorials. Accessed June 10, 2018. https://www.w3schools.com/.

"IOS 11: Swipe Left/right in UITableViewCell." Developers Log. June 28, 2017. Accessed June 10, 2018. https://developerslogblog.wordpress.com/2017/06/28/ios-11-swipe-leftright-in-uitableviewcell/.

"Resources for Developers, by Developers." MDN Web Docs. Accessed June 10, 2018. https://developer.mozilla.org/en-US/.

Sundell, John. "The Power of Switch Statements in Swift." Medium. September 24, 2017. Accessed June 10, 2018. https://medium.com/@johnsundell/the-power-of-switch-statements-in-swift-88516cb14aae.

"Token Based Authentication and HTTP/2 Example with APNS." Gobiko. September 24, 2016. Accessed June 10, 2018. http://gobiko.com/blog/token-based-authentication-http2-example-apns/.

Appendix - Hardware and Software Used

These are lists of hardware and software not created by me which are used in the project itself. Software used solely for development (i.e. Apple's Xcode, used for app development) is not included.

Hardware

- TP-Link Archer C7 AC1750 Router
- Raspberry Pi 3 (Model B)
- Apple iPhone X

Software

- LEDE 17.01 Operating System and Build System
  - Note: The LEDE project, which previously forked off of the OpenWrt project, is currently reintegrating with OpenWrt at the time of writing. This project is built using LEDE version 17.01, which is the most recent officially supported release of OpenWrt at this time.
- Raspbian 9 (Stretch)
- iOS 11 (and later)
- nodogsplash
- python
- Flask
- Gunicorn
- Hyper: HTTP Client
- Supervisord
- cron
- Bootstrap
- jQuery

Appendix - Selected Source Code and Configurations

Table of Contents

Captive Portal

Nodogsplash Configuration

```
GatewayInterface wlan1-1

FirewallRuleSet authenticated-users {
    FirewallRule block to 192.168.0.0/16
    FirewallRule block to 10.0.0.0/8
    FirewallRule allow tcp port 53
    FirewallRule allow udp port 53
    FirewallRule allow tcp port 80
    FirewallRule allow tcp port 443
    FirewallRule allow tcp port 22
}

FirewallRuleSet preauthenticated-users {
   FirewallRule allow tcp port 53
   FirewallRule allow udp port 53
   FirewallRule allow tcp port 5050 to <router-public-ip>
}

FirewallRuleSet users-to-router {
   FirewallRule allow udp port 53
   FirewallRule allow tcp port 53
   FirewallRule allow udp port 67
   FirewallRule allow tcp port 3222
}

EmptyRuleSetPolicy authenticated-users passthrough
EmptyRuleSetPolicy preauthenticated-users passthrough
EmptyRuleSetPolicy users-to-router block
EmptyRuleSetPolicy trusted-users passthrough
EmptyRuleSetPolicy trusted-users-to-router passthrough

GatewayName Blumenberg WiFi
ClientIdleTimeout 1440
ClientForceTimeout 10080
GatewayIPRange 10.0.2.0/24
```

Authentication Webpage

Nodogsplash Authentication Webpage

```
<html>
<head>
   <title>$gatewayname Entry</title>
   <meta HTTP-EQUIV="Pragma" CONTENT="no-cache">
   <!-- Pleace this snippet right after opening the head tag to make it work properly -->

   <!-- This code is licensed under GNU GPL v3 -->
   <!-- You are allowed to freely copy, distribute and use this code, but removing author credit is
strictly prohibited -->
   <!-- Generated by http://insider.zone/tools/client-side-url-redirect-generator/ -->

   <!-- REDIRECTING STARTS -->
   <link rel="canonical" href="https://oneb.ddns.net:5050/portal/$tok"/>
   <noscript>
         <meta http-equiv="refresh" content="0;URL=https://oneb.ddns.net:5050/portal/$tok">
   </noscript>
   <!--[if lt IE 9]><script type="text/javascript">var IE_fix=true;</script><![endif]-->
   <script type="text/javascript">
         var url = "https://oneb.ddns.net:5050/portal/$tok";
         if(typeof IE_fix != "undefined") // IE8 and lower fix to pass the http referer
         {
               document.write("redirecting..."); // Don't remove this line or appendChild() will fail
because it is called before document.onload to make the redirect as fast as possible. Nobody will see
this text, it is only a tech fix.
               var referLink = document.createElement("a");
               referLink.href = url;
               document.body.appendChild(referLink);
               referLink.click();
         }
         else { window.location.replace(url); } // All other browsers
   </script>
   <!-- Credit goes to http://insider.zone/ -->
   <!-- REDIRECTING ENDS -->
</head>
<body>
```

```
If you are not redirected automatically, please <a href="https://oneb.ddns.net:5050/portal/$tok">click
here</a> to access WiFi.

</body>
</html>
```

Authentication Webpage - Request Submission and Polling

```
function request() {
         if ($("#inputName").val() === "") {
            return;
         }

         $("#requestForm").hide();
         $("#requestButton").hide();
         $("#spinner").show();

         var xhttp = new XMLHttpRequest();
         xhttp.open("POST", "https://oneb.ddns.net:5050/req/" + $("#inputName").val(), false);
         xhttp.send();

         var requestId = xhttp.responseText;
         if (xhttp.status !== 200 || xhttp.responseText === "") {
            alert("Authorization request failed. Please try again or ask your host for more
information.");
            return;
         }

         function poll() {
            xhttp.open("GET", "https://oneb.ddns.net:5050/auth/" + requestId, false)
            xhttp.send();
            if (xhttp.status === 200) {
               window.location.replace("http://10.0.2.1:2050/nodogsplash_auth/?tok=$tok");
            } else if (xhttp.status === 202) {
               setTimeout(poll, 5 * 1000); //5 seconds between each poll
            } else if (xhttp.status === 403) {
               $("#message").append("Sorry, your access has been denied. Please ask your host for more
information.");
               $("#spinner").hide();
               $("#message").show();
            } else {
```

```
                $("#message").append("Something went wrong while checking the status of your
authorization. Please reload the page and try again or ask your host for more information. (" +
xhttp.status + ")");
                $("#spinner").hide();
                $("#message").show();
            }
        }

        setTimeout(poll, 10 * 1000); //wait 10 seconds initially
    }
```

Backend Server

Cleaner

```
import fcntl
import pickledb
import datetime

dbFile = open("database.db", "a+")
fcntl.flock(dbFile, fcntl.LOCK_EX)

db = pickledb.load("database.db", False)

count = 0
cutoff = (datetime.datetime.utcnow() - datetime.timedelta(minutes=10)).isoformat()
for entry in db.get('requests').keys():
    if db.get('requests').get(entry).get("timestamp") < cutoff:
        count += 1
        db.get('requests').pop(entry)

print("Cleaned " + str(count) + " stale requests")

db.dump()
fcntl.flock(dbFile, fcntl.LOCK_UN)
dbFile.close()
```

Rest Server - File Locking, Database Handling, and Open Requests

```
def loadDb():
    dbFile = open("database.db", "a+")
    fcntl.flock(dbFile, fcntl.LOCK_EX)
    db = pickledb.load("database.db", False)
    return dbFile, db


def closeDb(dbFile):
    fcntl.flock(dbFile, fcntl.LOCK_UN)
    dbFile.close()


def writeDb(dbFile, db):
    db.dump()
    closeDb(dbFile)


@app.route("/open", methods=['GET'])
def root():
    (dbFile, db) = loadDb()

    openRequests = (json.dumps({})
                    if db.get('requests') is None
                    else json.dumps(dict((requestId, db.get('requests').get(requestId))
                                         for requestId in db.get('requests').keys()
                                         if 'auth' not in db.get('requests').get(requestId))))

    closeDb(dbFile)
    return openRequests, 200
```

Rest Server - Sending Push Notifications

```
    apnToken = db.get("APNToken")
    if apnToken is not None:
        con = HTTPConnection('api.push.apple.co:443' if useProductionAPNServer else
'api.development.push.apple.com:443')
        method = 'POST'
        path = '/3/device/{0}'.format(apnToken)
        openRequests = 0 if db.get('requests') is None else len([requestId
                                                                 for requestId in db.get('requests').keys()
                                                                 if 'auth' not in
db.get('requests').get(requestId)])
        body = json.dumps({
            'aps': {
                'alert': "{0} is requesting WiFi access".format(name),
                'badge': openRequests,
                'sound': 'default'
            }
        })
        headers = {
            'apns-expiration': str(int(time.time()) + 600),
            'apns-priority': '10',
            'apns-topic': 'com.blumenberg.wifi.WiFi-Gate-Guard',
            'authorization': 'bearer {0}'.format(jwt.encode({
                                                     'iss': '26FA4P9HQ3',
                                                     'iat': time.time()
                                                 },
                                                 open('apnKey.p8').read(),
                                                 algorithm='ES256',
                                                 headers={
                                                     'alg': 'ES256',
                                                     'kid': 'NMA726AT3J'
                                                 }).decode('ascii'))
        }
        con.request(method, path, body, headers=headers)
        if con.get_response().status != 200:
            logFile = open("pushNotificationFailures.log", "a+")
            logFile.write("Error sending push notification ({0});
{1}\n".format(str(con.get_response().status),
                                                                                 con.get_response().read()))
            logFile.close()
```

Rest Server - Polling Response

```
@app.route("/auth/<string:requestId>", methods=['Get'])
def checkAuth(requestId):
    (dbFile, db) = loadDb()

    if requestId not in db.get('requests').keys():
        closeDb(dbFile)
        return "ID not found", 404

    if 'auth' in db.get('requests').get(requestId):
        if db.get('requests').get(requestId).get('auth'):
            db.get('requests').pop(requestId)
            writeDb(dbFile, db)
            return "Authorized", 200
        else:
            db.get('requests').pop(requestId)
            writeDb(dbFile, db)
            return "Unauthorized", 403
    else:
        closeDb(dbFile)
        return "Waiting for authorization", 202
```

Supervisord Configuration

```
[unix_http_server]
file=/tmp/supervisor.sock   ; the path to the socket file

[supervisord]
logfile=/var/log/supervisord.log ; main log file; default $CWD/supervisord.log
logfile_maxbytes=50MB        ; max main logfile bytes b4 rotation; default 50MB
logfile_backups=10           ; # of main logfile backups; 0 means none, default 10
loglevel=info                ; log level; default info; others: debug,warn,trace
pidfile=/tmp/supervisord.pid ; supervisord pidfile; default supervisord.pid
nodaemon=false               ; start in foreground if true; default false
minfds=1024                  ; min. avail startup file descriptors; default 1024
minprocs=200                 ; min. avail process descriptors;default 200

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
```

```
[supervisorctl]
serverurl=unix:///tmp/supervisor.sock ; use a unix:// URL  for a unix socket

[program:tokenRepository]
environment=PYTHONPATH=/usr/local/lib/python3.6/site-packages
command=/usr/local/bin/gunicorn --certfile /etc/letsencrypt/live/oneb.ddns.net/fullchain.pem --keyfile /
etc/letsencrypt/live/oneb.ddns.net/privkey.pem  -w 1 -b 10.0.0.115:5050 rest:app
directory=/www/server                  ; the directory where the server's python code is
```

App

Keeping Request List Updated

```
func periodicUpdate() {
    if let nextUpdate = nextUpdate {
        switch Date().compare(nextUpdate) {
            case .orderedSame: fallthrough
            case .orderedDescending: //nextUpdate is before now, we should update
                Request.update() {
                    success in
                    if success {
                        self.reloadData()
                        self.nextUpdate = Date().addingTimeInterval(15)
                    }
                    DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(15)) {
                        self.periodicUpdate()
                    }
                }
            case .orderedAscending:
                // nextUpdate is after now, but the last asyncAfter call is running, which means the
                // user manually updated. We should just reset the timer for another 15 seconds
                DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(15)) {
                    self.periodicUpdate()
                }
        }
    }
}
```

```
override func viewDidLoad() {
    super.viewDidLoad()
    Request.update() {
        success in
        if success {
            self.reloadData()
            self.nextUpdate = Date().addingTimeInterval(15)
            if let refreshControl = self.refreshControl, refreshControl.isRefreshing {
                refreshControl.endRefreshing()
            }
        }
        DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(15)) {
            self.periodicUpdate()
        }
    }
}

@IBAction func refresh(_ sender: UIRefreshControl) {
    Request.update() {
        success in
        if success {
            self.reloadData(shouldShowActions: false)
            sender.endRefreshing()
            self.nextUpdate = Date().addingTimeInterval(15)
        }
    }
}
```

Peek Animation

```
func showActions(forRow row:Int) {
    guard let cell = (tableView.cellForRow(at: IndexPath(row: row, section: 0))) as?
RequestTableViewCell else {
        return
    }

    let labelWidth:CGFloat = 20

    let createLabel:(UIColor)->UILabel = {
        color in
```

```
        let label = UILabel(frame: CGRect.zero)
        label.backgroundColor = color
        label.clipsToBounds = false
        label.frame = CGRect(x: cell.bounds.width, y: 0, width: labelWidth, height: cell.bounds.height)
        return label
    }

    let greenLabel = createLabel(.green)
    let redLabel = createLabel(.red)

    //ordering of the subviews is key to get it to look right
    cell.insertSubview(greenLabel, aboveSubview: cell.contentView)
    cell.insertSubview(redLabel, belowSubview: greenLabel)

    UIView.animate(withDuration: 0.5, delay: 0, usingSpringWithDamping: 1, initialSpringVelocity: 3,
options: .curveEaseOut, animations: {
        cell.nameLabelLeadingConstraint.constant -= (labelWidth * 2)
        cell.requestAgeLabelLeadingConstraint.constant -= (labelWidth * 2)
        cell.layoutIfNeeded()

        greenLabel.transform = greenLabel.transform.translatedBy(x: -(labelWidth), y: 0)
        redLabel.transform = redLabel.transform.translatedBy(x: -(labelWidth * 2), y: 0)
    }, completion: {
        _ in
        UIView.animate(withDuration: 0.5, delay: 0, usingSpringWithDamping: 1, initialSpringVelocity: 3,
options: .curveEaseOut, animations: {
            cell.nameLabelLeadingConstraint.constant += (labelWidth * 2)
            cell.requestAgeLabelLeadingConstraint.constant += (labelWidth * 2)
            cell.layoutIfNeeded()

            greenLabel.transform = CGAffineTransform.identity
            redLabel.transform = CGAffineTransform.identity
        }, completion: {
            _ in
            greenLabel.removeFromSuperview()
            redLabel.removeFromSuperview()
        })
    })
}
```

Registering for Push Notifications

```
static func sendAPNTokenToServer(vc:UIViewController?) {
    if let apnToken = UserDefaults.standard.string(forKey: "APNToken") {
        var urlComponents = URLComponents()
        urlComponents.scheme = "https"
        urlComponents.host = "oneb.ddns.net"
        urlComponents.port = 5050
        urlComponents.path = "/registerAPNToken/\(apnToken)"
        guard let url = urlComponents.url else {
            fatalError("Failed to create url to send apn token.")
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        URLSession(configuration: .default).dataTask(with: request) {
            data, response, error in
            DispatchQueue.main.async {
                if let vc = vc {
                    if let error = error {
                        let alert = UIAlertController(title: "Failed to register with token server for
push notifications", message: error.localizedDescription, preferredStyle: .alert)
                        alert.addAction(UIAlertAction(title: "OK", style: .default))
                        vc.present(alert, animated: true, completion: nil)
                    } else {
                        let alert = UIAlertController(title: "Successfully registered for push
notifications", message: nil, preferredStyle: .alert)
                        alert.addAction(UIAlertAction(title: "Great!", style: .default))
                        vc.present(alert, animated: true, completion: nil)
                    }
                }
            }
        }.resume()
    } else if let vc = vc {
        let alert = UIAlertController(title: "No apn token saved in user defaults", message: nil,
preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .default))
        vc.present(alert, animated: true, completion: nil)
    }
}
```

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    if #available(iOS 10, *) {
        UNUserNotificationCenter.current().requestAuthorization(options: [.badge, .alert]) {
            success, error in
            DispatchQueue.main.async {
                if success {
                    application.registerForRemoteNotifications()
                } else if let error = error, let vc = self.window?.rootViewController {
                    let alert = UIAlertController(title: "Failed to get authorization for push
notifications", message: error.localizedDescription, preferredStyle: .alert)
                    alert.addAction(UIAlertAction(title: "OK", style: .default))
                    vc.present(alert, animated: true, completion: nil)
                }
            }
        }
    }

    return true
}

func application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken
deviceToken: Data) {
    let deviceTokenString = deviceToken.reduce("", {$0 + String(format: "%02X", $1)})
    let oldDeviceTokenString = UserDefaults.standard.string(forKey: "APNToken")

    if oldDeviceTokenString == nil || oldDeviceTokenString != deviceTokenString {
        UserDefaults.standard.set(deviceTokenString, forKey: "APNToken")
        sendAPNTokenToServer(vc: self.window?.rootViewController)
    }
}

func application(_ application: UIApplication, didFailToRegisterForRemoteNotificationsWithError error:
Error) {
    if let vc = window?.rootViewController  {
        let alert = UIAlertController(title: "Failed to register for push notifications", message:
error.localizedDescription, preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .default))
        vc.present(alert, animated: true, completion: nil)
    }
}
```